LITTLE Newsletter #10 April 2, 1972

Interspersing Macros J. T. Schwartz

a. Statement of Problem:

An expressional technique useful in a variety of programming stituations is the segregation of source text into two portions: a main part expressing the 'principal' logic of an algorithm and one or more subsidiary parts indicating 'auxiliary' actions to be taken in connection with particular items belonging to the main part. The point here is that by isolating the logical skeleton of a process we make its 'main idea' stand out and allow ourselves to express this main idea without forcing 'details' to be stated at the same time. The comprehension of details subsequently supplied is also made easier, since they appear in a pre-specified context.

The type of situation I have in mind is best made by considering an example.

A good example is furnished by the expanded Backus top-down recognizer metalanguage described on pp. 75-90 of the Cocke-Schwartz compiler notes. Describing a top-down parser always involves three logically separable tasks:

a. specification of the sequence of syntactic elements to be sought in an input string (the skeletal 'Backus grammar' of the language);

b. for each syntactic element appearing in a sequence a), a directive concerning the action to be taken if the element is missing or ill formed in the input string being parsed;

c. specification of the 'generative' actions to be taken upon complete recognition of each of the component parts of a syntactic string.

Pars a) and b) together describe the 'branching structure' of a top-down parsing process; c) describes the collection of 'procedural elements' attached to this

branching structure. It is not unreasonable to write a) first and c) second. Most of the entries in b) describe an error-action to be on the failure of a test in a), and it is therefore reasonable to write b) after a) and c) have been completed. Note in particular that if a) and b) are intermixed one gets sequences having the following flavor:

"if testl succeeds do actl; if it fails take transfer 1;
if text 2 succeeds do act 2; if it fails take transfer2,...".

Seen 'en masse', expressions of this kind tend to be confusing. This same point is made if we observe that the footnoted Backus description

a) <forstatement> := FØR <*name>'=' <expr> TØ <expr>STEP<expr> [*]

c) [*]: call forstaement generator; begin processing of next statement

b) try next statement form; emit error message 1;
emit error message 2; emit error message 3;
emit error message 4, emit error message 5,
go to <shortforstatement>, emit error message 6

is distinctly cleaner than the interspersed text which it implies:

<forstatement> := FOR [if missing try next statement form] <*name>
[if missing emit error message 1] '=' [if missing emit error message 2] ... etc.

b. A helpful mechanism, suggested as a partial solution:

As a mechanism allowing textual 'footnoting' of the kind which the above example suggests as useful, the following is proposed. Introduce argument-free 'interspersing macros'. Using a style like that already used in the LITTLE macro-producer, we may agree that to define such a macro one writes

+*<integer> = textl ** text2 ** ... ** textn *** .

Example:

+* 1 = go to 1 ** call er(1) ** call er(2) ** call er(3)
** call er(4) ***

Note that the macro-definition shown above is terminated by the occurrence of three rather than of two asterisks; the asterisk-pairs which occur separate the several 'successive definitions' of the macro, see below. An interspersing macro defined in this way is called by writing /<integer>/, e.g. /1/. Each time the macro is called, one of the text fragments textl, text2, ... appearing in its definition replaces its call. These replacements are made successively, i.e., textl on the first call, text2 on the second, etc. The occurrence of more calls than there are text fragments in the macro's definition is an error, and yields an error message, though of course a macro redefinition is always possible, and renews the sequence of text fragments corresponding to a given <integer>.

In many cases, one will wish to verify that all the text fragments appearing in an interspersing-macro definition have been called by the time that a given point in a source text is reached. The following supplementary mechanism will allow this: introduce a 'post-call' having the form /<integer>-/, e.g. /1-/. This simply generates a standard error message if there still remain unissued text fragments for the interspersing macro to which the post-call refers.

Interspersing macros of the form suggested can be used effectively in connection with ordinary macros. In particular, expansion of an interspersing macro can produce ordinary macro-calls and vice-versa. Note in

particular that the syntax of the 'extended Backus' metalanguage discussed above can be captured pretty closely by a combination of ordinary and interspersing macros. For example, the extended Backus (cf. Cocke-Schwartz, p. 95)

```
(A) <FØRSTAT> = FØR <*NAME>'=' <EXPR> TØ <EXPR> STEP
                    <EXPR>.GENFØR..CDEND/.B.ER(4).ER(5).ER(1).
                      .ER(11)..ER(1)
                = ...
```

can be written as

```
(B)  +*1 = B ** ER(4) ** ER(5) ** ER(1) ** ER(11) ** I N1**ER(1)***
    /FØRSTAT/ R('FØR') F(NAME) R('=') S(EXPR) R('TØ')
              S(EXPR)R('STEP')S(EXPR)C GENFØR., G CDEND.,
    /N1/  ... ...
```

Here we have assumed the following overall macros to be in effect:

```
     +* R(X) = CALL RECØG(X)., /1/ **
     +* F(X) = CALL FIND(X)., /1/ **
     +* S(X) = CALL SUBPART(X)., /1/ **
     +* C    = CALL **        +* G = GØ TØ **  .
```

The text (B) is not too bad a substitute for the slightly better (A).